COMP 382: Reasoning about Algorithms

Greedy Algorithms: Huffman Encoding, Caching Problem

Prof. Maryam Aliakbarpour

co-instructors: Prof. Anjum Chida & Prof. Konstantinos Mamouras

October 21, 2025

Today's Lecture

1. **Huffman Encoding**

2. **The Caching Problem**

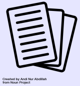
Reading:

- Chapter 5 of the Algorithms book [Dasgupta et al., 2006]
- Chapter 14 of [Roughgarden, 2022]
- Chapter 4.3 of [Tardos and Kleinberg, 2005]

1. Huffman Encoding

A greedy algorithm form string compression

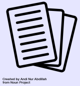
The World in Zeros and Ones



Storing a File

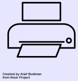
Characters in a file
are saved on a disk
as their binary code
equivalents (e.g., ASCII).

The World in Zeros and Ones



Storing a File

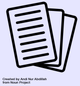
Characters in a file are saved on a disk as their binary code equivalents (e.g., ASCII).



Sending Text to a Printer

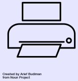
Text is sent to a printer as a sequence of binary data, which the printer interprets.

The World in Zeros and Ones



Storing a File

Characters in a file are saved on a disk as their binary code equivalents (e.g., ASCII).



Sending Text to a Printer

Text is sent to a printer as a sequence of binary data, which the printer interprets.

Digital Audio (MP3)

Sound is digitized into numbers, then encoded into a compressed format like MP3.

Encoding for Digitalization

An **encoding** is a scheme that maps a message (a sequence of characters) from an alphabet into a message in another alphabet, most commonly binary digits (bits).

Example: **ASCII** (American Standard Code for Information Interchange) It assigns a unique number (and thus a unique binary code) to every letter, digit, and punctuation mark.

Character	$\rightarrow$	**8-Bit Binary Code**
a	$\rightarrow$	01100001
b	$\rightarrow$	01100010
:	$\rightarrow$	00111010
1	$\rightarrow$	00110001
...	$\rightarrow$	...

Encoding for Digitalization

An **encoding** is a scheme that maps a message (a sequence of characters) from an alphabet into a message in another alphabet, most commonly binary digits (bits).

Example: **ASCII** (American Standard Code for Information Interchange) It assigns a unique number (and thus a unique binary code) to every letter, digit, and punctuation mark.

Code

Character	$\rightarrow$	8-Bit Binary Code	
a	$\rightarrow$	01100001	Codeword
b	$\rightarrow$	01100010	
:	$\rightarrow$	00111010	
1	$\rightarrow$	00110001	
...	$\rightarrow$	...	

Encoding and Decoding

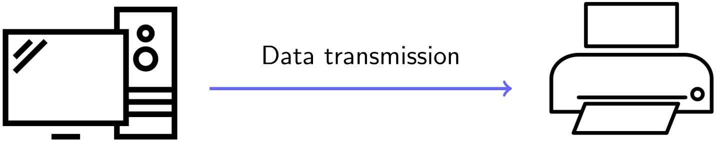
Example: Consider a string with four symbols: A, B, C, and D, and a 2-bit code for each symbol

$$\text{A}\rightarrow 00 \qquad \text{B}\rightarrow 01 \qquad \text{C}\rightarrow 10 \qquad \text{D}\rightarrow 11$$

Encoding and Decoding

Example: Consider a string with four symbols: A, B, C, and D, and a 2-bit code for each symbol

$$\text{A}\rightarrow 00 \qquad \text{B}\rightarrow 01 \qquad \text{C}\rightarrow 10 \qquad \text{D}\rightarrow 11$$



Created by Suharsono
from Noun Project

Created by Arief Budiman
from Noun Project

Encoding

$$\texttt{ADB} \rightarrow \underbrace{\texttt{00}}\,\underbrace{\texttt{11}}\,\underbrace{\texttt{01}}$$

Decoding

$$\underbrace{\texttt{00}}\,\underbrace{\texttt{11}}\,\underbrace{\texttt{01}} \rightarrow \texttt{ADB}$$

Fixed Length Code

A **fixed length code** uses the same number of bits for each symbol.

Example: If we use this 2-bit code to encode a string of length 130 million bits, this fixed length code requires:

$$130 \text{ million symbols} \times 2 \text{ bits/symbol} = \textbf{260 million bits}.$$

But what if the symbols appear with different frequencies?

Symbol	**Frequency**
A	70 million
B	3 million
C	20 million
D	37 million

Variable Length Code

Idea: Can we use shorter codewords for frequent symbols (like A) and longer ones for infrequent symbols (like B)?

We can have a **variable-length code**: A=0, B=001, C=11, D=01.

This variable length code requires:

- A (1 bit): $70\ mil. \times 1 = 70$ million bits
- B (3 bits): $3\ mil. \times 3 = 9$ million bits
- C (2 bits): $20\ mil \times 2 = 40$ million bits
- D (2 bits): $37\ mil \times 2 = 74$ million bits

---

193 million bits

Variable Length Code

Idea: Can we use shorter codewords for frequent symbols (like A) and longer ones for infrequent symbols (like B)?

We can have a **variable-length code**: A=0, B=001, C=11, D=01.

This variable length code requires:

- A (1 bit): 70 $mil. \times 1 =$ 70 million bits
- B (3 bits): 3 $mil. \times 3 =$ 9 million bits
- C (2 bits): 20 $mil \times 2 =$ 40 million bits
- D (2 bits): 37 $mil \times 2 =$ 74 million bits

---

193 million bits $\downarrow$ %25 reduction in the size

The Fundamental Question

What is the most efficient way to encode a string?

The Fundamental Question

What is the most efficient way to encode a string?

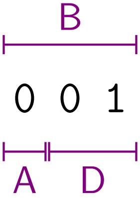
- What properties must a code have to be instantly and unambiguously decodable?
- Given the frequency of each symbol, find the valid code that produces the shortest possible encoded message.

The Prefix-Free Property

The Challenge: Variable-length codes can be ambiguous. In our example, the bit-string 001 is undecipherable. It could be AD or B.

The Prefix-Free Property

The Challenge: Variable-length codes can be ambiguous. In our example, the bit-string 001 is undecipherable. It could be AD or B. This happens because the code for A is a prefix of the code for B:



The Prefix-Free Property

The Challenge: Variable-length codes can be ambiguous. In our example, the bit-string 001 is undecipherable. It could be AD or B. This happens because the code for A is a prefix of the code for B:

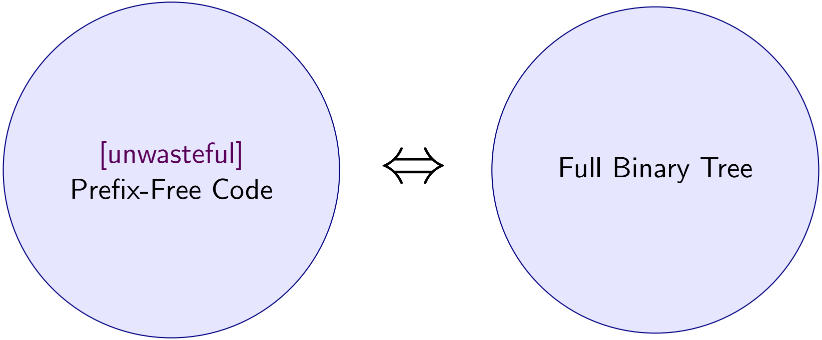
$$\overbrace{0\ 0\ 1}^{B}$$
$$\underbrace{0}_{A}\ \underbrace{0\ 1}_{D}$$

The Solution: Prefix-Free Codes!

We ensure that our encoding follows this rule: No codeword is a prefix of any other codeword. This property ensures that any encoded string is uniquely decipherable.

Binary Tree Representation

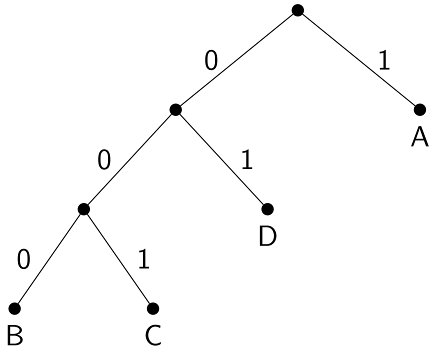
Any prefix-free code can be represented by a **full binary tree** (where every internal node has two children).



Binary Tree Representation

How it works:

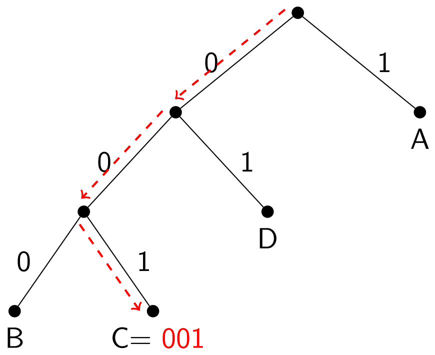
- The symbols (A, B, C, D) are the leaves of the tree.



Binary Tree Representation

How it works:

- The symbols (A, B, C, D) are the leaves of the tree.
- The path from the root to a leaf generates its codeword, using 0 for a left branch and 1 for a right branch.

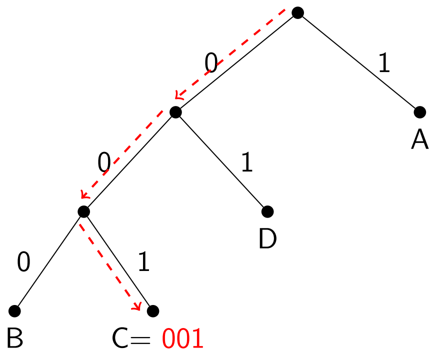


Binary Tree Representation

How it works:

- The symbols (A, B, C, D) are the leaves of the tree.
- The path from the root to a leaf generates its codeword, using 0 for a left branch and 1 for a right branch.

Decoding: Start at the root and follow the path based on the input bits. When you reach a leaf, output the symbol and return to the root.



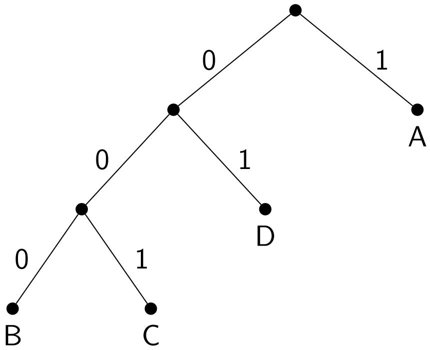
Binary Tree Representation

- **Why is this code prefix-free?**

  If a symbol was at an internal node, it would be a prefix for all of its successors. Having the symbols in the leaves implies that no codeword is the prefix of another one.

- **Why do we consider only full binary trees?**

  If a node has only a single child, we can improve the code by removing that node. This change shortens the codeword's length without causing any issues.



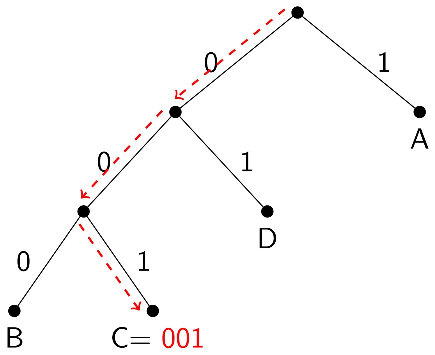
Binary Tree Representation

- **Why is this code prefix-free?**

  If a symbol was at an internal node, it would be a prefix for all of its successors. Having the symbols in the leaves implies that no codeword is the prefix of another one.

- **Why do we consider only full binary trees?**

  If a node has only a single child, we can improve the code by removing that node. This change shortens the codeword's length without causing any issues.



The Fundamental Question

What is the most efficient way to encode a string?

$$=$$

What is the optimal prefix-free code/binary tree that minimizes the message length based on symbol frequencies?

The Fundamental Question

What is the most efficient way to encode a string?

$$=$$

What is the optimal prefix-free code/binary tree that minimizes the message length based on symbol frequencies?

We will now explore Huffman's algorithm, a classic and elegant case where a simple greedy strategy **is provably optimal** for solving the prefix-free data compression problem.

Defining Cost of a Tree

Suppose we have n symbols in alphabet Σ, and the f_a denotes the frequency of element a.

Defining Cost of a Tree

Suppose we have n symbols in alphabet Σ, and the f_a denotes the frequency of element a.

The **cost of a tree**, denoted by T, is the total length of the encoded message, which can be expressed in terms of the weighted sum of the path lengths.

$$\text{Cost:} \qquad L(T) := \sum_{x \in \Sigma} f_x \times (\text{length of codeword for symbol } x)$$
$$= \sum_{x \in \Sigma} f_x \times (\text{depth of symbol } x)$$

Huffman's Greedy Algorithm

The core insight: The two symbols with the smallest frequencies must be siblings at the lowest level of the optimal tree.

The Greedy Strategy

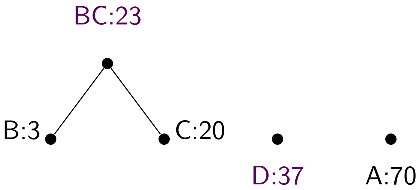
1. Identify the two symbols with the lowest frequencies.
2. Join them as children of a new parent node. This parent's frequency is the sum of its children's frequencies $(f_i + f_j)$.
3. Remove the original two symbols from the list and add this new parent node.
4. Repeat this process until only one node remains—the root of the tree.

Huffman Algorithm: An Example

B:3 C:20 D:37 A:70

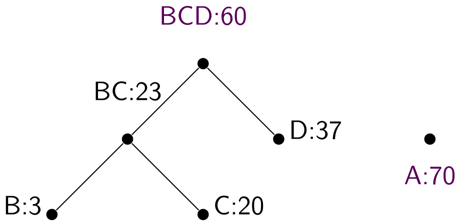
Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

Huffman Algorithm: An Example



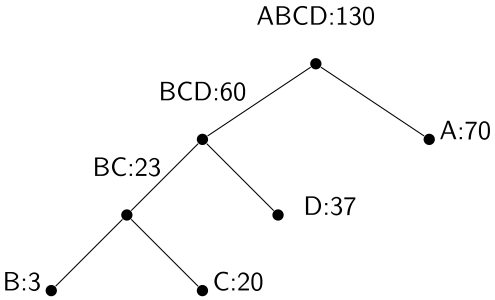
Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

Huffman Algorithm: An Example



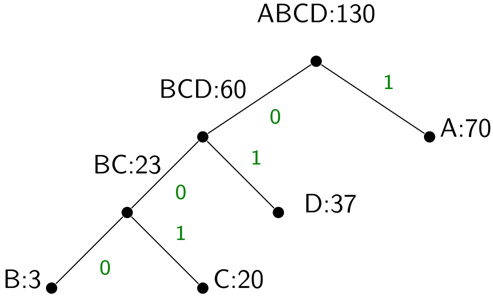
Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

Huffman Algorithm: An Example



Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

Huffman Algorithm: An Example

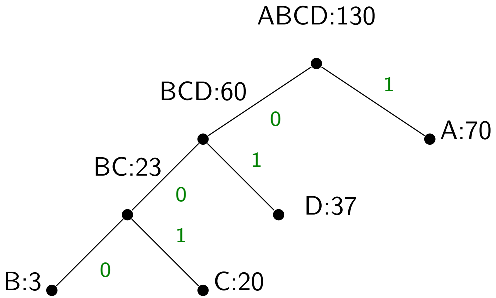


Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

The Huffman algorithm gives us the following code:

$$A \to 1 \qquad B \to 000 \qquad C \to 001 \qquad D \to 01$$

Huffman Algorithm: An Example



Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

The Huffman algorithm gives us the following code:

$$A \to 1 \qquad B \to 000 \qquad C \to 001 \qquad D \to 01$$

Total length: 213 million bits $\downarrow$ %18 reduction in the size

The Algorithm in Pseudocode

Input: An array ‘f‘ of frequencies for ‘n‘ symbols.
Data Structure: Use a priority queue ‘H‘ to find minimums.

procedure Huffman(f)

1. **Initialize:** Insert all ‘n‘ symbols into the priority queue ‘H‘.
2. **Iterate** $n-1$ **times** (from $k = n+1$ to $2n-1$):
   - Extract the two nodes with the minimum frequencies:
     `i = deletemin(H)`, `j = deletemin(H)`.
   - Create a new parent node ‘k‘ with children ‘i‘ and ‘j‘.
   - Set the new node’s frequency: `f[k] = f[i] + f[j]`.
   - Insert the new node ‘k‘ back into ‘H‘.

The algorithm’s runtime is $O(n \log n)$.

Proof of Optimality

Theorem

For every alphabet Σ and non-negative symbol frequencies $\{f_x\}_{x \in \Sigma}$, the Huffman algorithm outputs a prefix-free code with the minimum-possible encoding length.

Proof of Optimality

Theorem

For every alphabet Σ and non-negative symbol frequencies $\{f_x\}_{x \in \Sigma}$, the Huffman algorithm outputs a prefix-free code with the minimum-possible encoding length.

In other words, the algorithm finds a full binary tree (a Σ-tree) with the minimum possible weighted leaf depth, where the average is weighted by symbol frequencies.

$$L(T) = \sum_{x \in \Sigma} f_x \cdot (\text{depth of leaf } x \text{ in } T)$$

Proof by Induction

- **Statement:**

  $P(n) \leftarrow$ "The Huffman algorithm is correct for any alphabet of size at most n."

Proof by Induction

- **Statement:**

  $P(n) \leftarrow$ "The Huffman algorithm is correct for any alphabet of size at most n."

- **Base Case:** $P(2)$ For an alphabet with two symbols, Huffman's algorithm assigns one symbol the code '0' and the other '1'. This is trivially optimal.

Proof by Induction

- **Statement:**

  $P(n) \leftarrow$ "The Huffman algorithm is correct for any alphabet of size at most n."

- **Base Case:** $P(2)$ For an alphabet with two symbols, Huffman's algorithm assigns one symbol the code '0' and the other '1'. This is trivially optimal.

- **Inductive Hypothesis:** Assume $P(n-1)$ is true for $n > 2$. That is, assume Huffman's algorithm is correct for all alphabets with size less than n.

Proof by Induction

- **Statement:**

  $P(n) \leftarrow$ "The Huffman algorithm is correct for any alphabet of size at most n."

- **Base Case:** $P(2)$ For an alphabet with two symbols, Huffman's algorithm assigns one symbol the code '0' and the other '1'. This is trivially optimal.

- **Inductive Hypothesis:** Assume $P(n-1)$ is true for $n > 2$. That is, assume Huffman's algorithm is correct for all alphabets with size less than n.

Created by Jamie Dickinson
from Noun Project

- **Inductive Step:** We must prove $P(n)$ is true, using the inductive hypothesis.

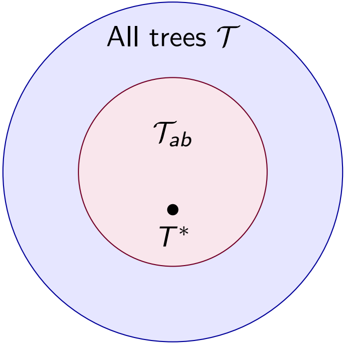
Proof Strategy: High-Level Plan

Let a and b be the symbols with the two smallest frequencies. The proof hinges on two main ideas.

- **Claim #1** guarantees that Huffman's algorithm finds the best possible tree among the set of trees where a and b are siblings.
- **Claim #2 (Exchange Argument)** guarantees that there is an optimal tree where the two lowest-frequency symbols, a and b, are siblings.
- Combining these two ideas, the tree produced by Huffman's algorithm must be an optimal tree.

Proof Strategy: High-Level Plan

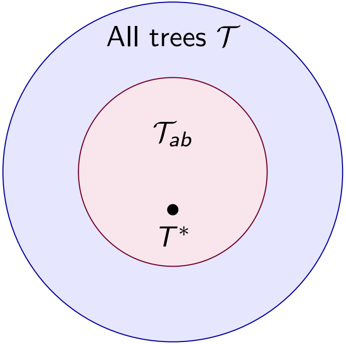
- **Goal:** Prove Huffman's algorithm finds the optimal tree in the set of all trees $\mathcal{T}$ (the outer circle).



$\mathcal{T} \leftarrow$ all possible trees
$\mathcal{T}_{ab} \leftarrow$ trees where a and b as siblings.

Proof Strategy: High-Level Plan

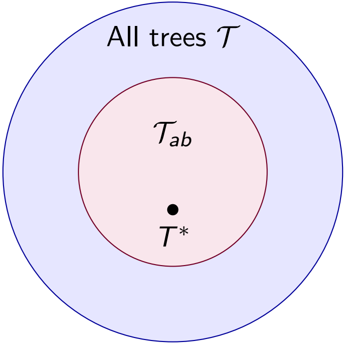
- **Goal:** Prove Huffman's algorithm finds the optimal tree in the set of all trees $\mathcal{T}$ (the outer circle).
- **Claim #1:** The algorithm finds the optimal tree within the restricted set $\mathcal{T}_{ab}$ (the inner circle).



$\mathcal{T} \leftarrow$ all possible trees
$\mathcal{T}_{ab} \leftarrow$ trees where a and b as siblings.

Proof Strategy: High-Level Plan

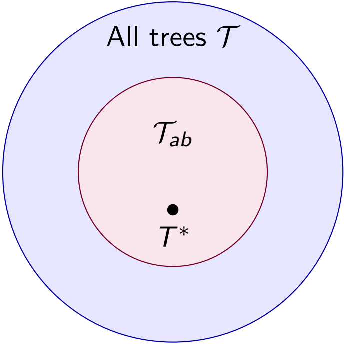
- **Goal:** Prove Huffman's algorithm finds the optimal tree in the set of all trees $\mathcal{T}$ (the outer circle).
- **Claim #1:** The algorithm finds the optimal tree within the restricted set $\mathcal{T}_{ab}$ (the inner circle).
- **Claim #2:** An optimal tree from $\mathcal{T}$ is guaranteed to also be in $\mathcal{T}_{ab}$.



$\mathcal{T} \leftarrow$ all possible trees
$\mathcal{T}_{ab} \leftarrow$ trees where a and b as siblings.

Proof Strategy: High-Level Plan

- **Goal:** Prove Huffman's algorithm finds the optimal tree in the set of all trees $\mathcal{T}$ (the outer circle).
- **Claim #1:** The algorithm finds the optimal tree within the restricted set $\mathcal{T}_{ab}$ (the inner circle).
- **Claim #2:** An optimal tree from $\mathcal{T}$ is guaranteed to also be in $\mathcal{T}_{ab}$.
- **Conclusion:** Therefore, finding the optimum in the inner circle is sufficient to find the global optimum.



$\mathcal{T} \leftarrow$ all possible trees
$\mathcal{T}_{ab} \leftarrow$ trees where a and b as siblings.

Claim #1: Optimality in a Restricted Set

Lemma (Optimality in $\mathcal{T}_{ab}$)

Suppose we have an alphabet Σ of size n. If Huffman's algorithm outputs an optimal tree for any alphabet of size $n-1$, then the output of the Huffman algorithm minimizes the weighted leaf depth over all Σ-trees in which a and b are siblings.

Proving Claim #1: Correspondence

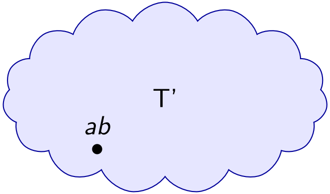
Consider a new alphabet Σ' where we "fuse" a and b into a new pseudo-symbol 'ab'.

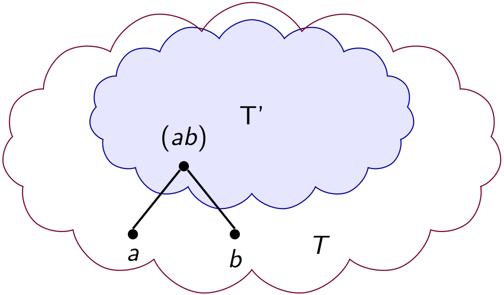
- The frequency of 'ab' is $f_{ab} = f_a + f_b$.
- $\Sigma' = (\Sigma \setminus \{a, b\}) \cup \{ab\}$
- $|\Sigma'| = n - 1$

Proving Claim #1: Correspondence

Consider a new alphabet Σ' where we "fuse" a and b into a new pseudo-symbol 'ab'.

- The frequency of 'ab' is $f_{ab} = f_a + f_b$.
- $\Sigma' = (\Sigma \setminus \{a, b\}) \cup \{ab\}$
- $|\Sigma'| = n - 1$





Proving Claim #1: Preserving Cost

Let T' be a Σ'-tree and T be the corresponding tree in $\mathcal{T}_{ab}$. The weighted leaf depths of T and T' are related by:

$$L(T) = L(T') + f_a + f_b$$

Proving Claim #1: Preserving Cost

Let T' be a Σ'-tree and T be the corresponding tree in $\mathcal{T}_{ab}$. The weighted leaf depths of T and T' are related by:

$$L(T) = L(T') + f_a + f_b$$

Proof: For any symbol $x \neq a, b, ab$, its depth is the same in T and T'. The leaves for a and b in T are one level deeper than the leaf for 'ab' in T'.

$$\begin{aligned} L(T) - L(T') &= f_a \cdot d_T(a) \qquad\qquad + f_b \cdot d_T(b) \qquad\qquad - f_{ab} \cdot d_{T'}(ab) \\ &= f_a \cdot (d_{T'}(ab) + 1) + f_b \cdot (d_{T'}(ab) + 1) - (f_a + f_b) \cdot d_{T'}(ab) \\ &= f_a + f_b \end{aligned}$$

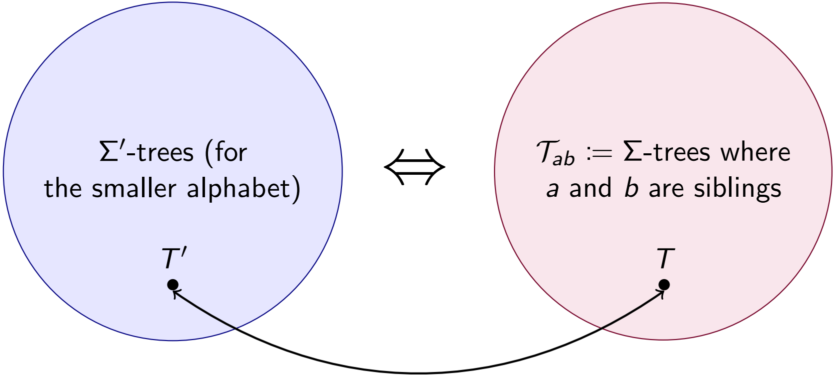
Proving Claim #1: The Correspondence

There is a one-to-one correspondence between:

Σ'-trees (for the smaller alphabet) $\Longleftrightarrow$ $\mathcal{T}_{ab} :=$ Σ-trees where a and b are siblings

Proving Claim #1: The Correspondence

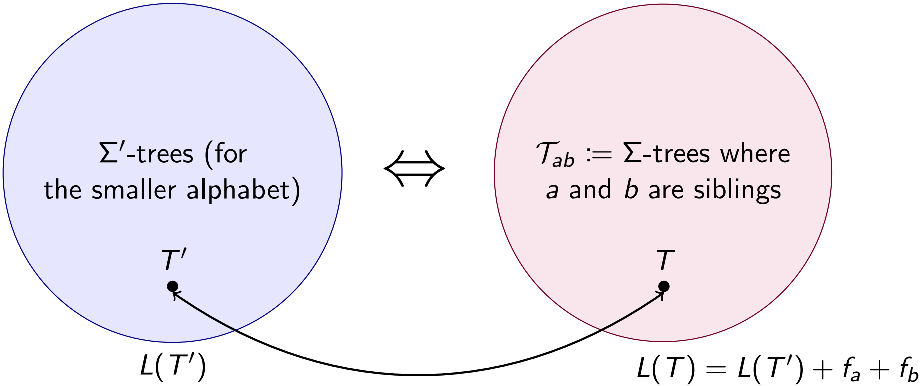
There is a one-to-one correspondence between:



Since $f_a + f_b$ is a constant, a tree T is optimal in $\mathcal{T}_{ab}$ if and only if the corresponding T' is an optimal Σ'-tree.

Proving Claim #1: The Correspondence

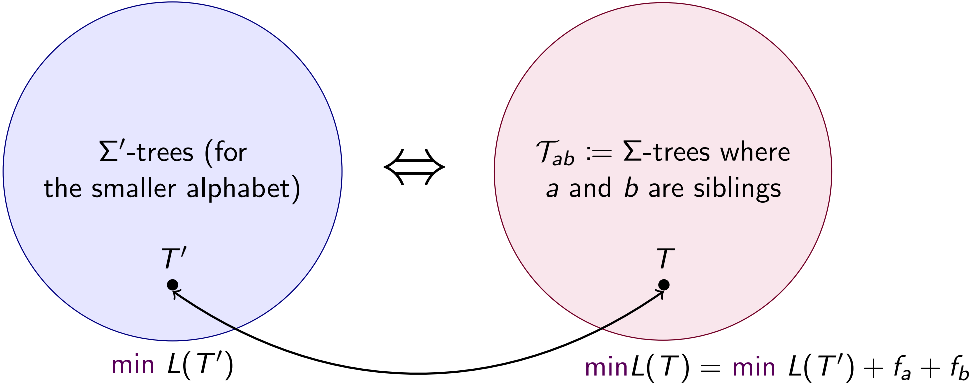
There is a one-to-one correspondence between:



Since $f_a + f_b$ is a constant, a tree T is optimal in $\mathcal{T}_{ab}$ if and only if the corresponding T' is an optimal Σ'-tree.

Proving Claim #1: The Correspondence

There is a one-to-one correspondence between:



Since $f_a + f_b$ is a constant, a tree T is optimal in $\mathcal{T}_{ab}$ if and only if the corresponding T' is an optimal Σ'-tree.

Proving Claim #1: Putting It All Together

We can now connect the pieces to prove Main Idea $\#1$:

1. Huffman's algorithm on Σ begins by merging a and b. The rest is equivalent to running it on the *residual problem* (Σ'), so T_{Huff} lies in $\mathcal{T}_{ab}$ and corresponds to a residual tree T'_{Huff} on Σ'.

Proving Claim #1: Putting It All Together

We can now connect the pieces to prove Main Idea $\#1$:

1. Huffman's algorithm on Σ begins by merging a and b. The rest is equivalent to running it on the *residual problem* (Σ'), so T_{Huff} lies in $\mathcal{T}_{ab}$ and corresponds to a residual tree T'_{Huff} on Σ'.
2. By the **inductive hypothesis** (since $|\Sigma'| = k - 1$), T'_{Huff} is an optimal Σ'-tree.

Proving Claim #1: Putting It All Together

We can now connect the pieces to prove Main Idea $\#1$:

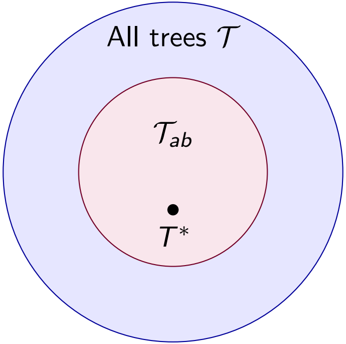
1. Huffman's algorithm on Σ begins by merging a and b. The rest is equivalent to running it on the *residual problem* (Σ'), so T_{Huff} lies in $\mathcal{T}_{ab}$ and corresponds to a residual tree T'_{Huff} on Σ'.
2. By the **inductive hypothesis** (since $|\Sigma'| = k - 1$), T'_{Huff} is an optimal Σ'-tree.
3. Because the cost mapping preserves optimality, T_{Huff} must be the optimal tree among all trees in $\mathcal{T}_{ab}$.

Thus, Huffman's algorithm is optimal over the set $\mathcal{T}_{ab}$, which implies Claim #1.

Claim #2: An Optimal solution exists in restricted set

Lemma

There is an optimal tree where the two lowest-frequency symbols, a and b, are siblings.



Proving Claim #2: The Exchange Argument

We now prove that an optimal tree must exist in $\mathcal{T}_{ab}$.

Let T^* be an arbitrary optimal Σ-tree. Let x and y be two symbols that are siblings at the deepest level of T^*.

If $\{a, b\} = \{x, y\}$, we are done.

Proving Claim #2: The Exchange Argument

We now prove that an optimal tree must exist in $\mathcal{T}_{ab}$.

Let T^* be an arbitrary optimal Σ-tree. Let x and y be two symbols that are siblings at the deepest level of T^*.

If $\{a, b\} = \{x, y\}$, we are done.

If not, we create a new tree $\tilde{T}$ by swapping the positions of a and x.

Proving Claim #2: The Exchange Argument

We now prove that an optimal tree must exist in $\mathcal{T}_{ab}$.

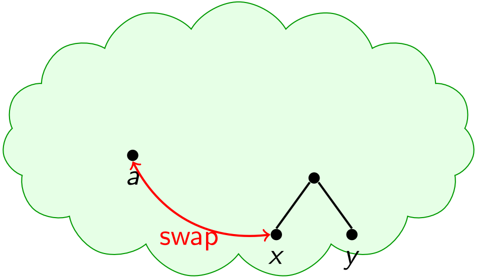
Let T^* be an arbitrary optimal Σ-tree. Let x and y be two symbols that are siblings at the deepest level of T^*.

If $\{a, b\} = \{x, y\}$, we are done.

If not, we create a new tree $\tilde{T}$ by swapping the positions of a and x.

Let's compare the cost:

$$
\begin{aligned}
L(T^*) - L(\tilde{T}) \\
&= (f_x - f_a) \cdot (\mathrm{d}_{T^*}(x) - \mathrm{d}_{T^*}(a)) \\
&\geq 0
\end{aligned}
$$



T^* : An optimal tree

Proving Claim #2: The Math

As long as $f_a \leq f_x$, by swapping leaf a with a leaf x at the deepest level, the cost of the tree does not increase.

Due to this fact we can swap a (the lowest frequency element) with x (the leaf at the deepest level), without increasing the cost of the tree.

Proving Claim #2: The Math

As long as $f_a \leq f_x$, by swapping leaf a with a leaf x at the deepest level, the cost of the tree does not increase.

Due to this fact we can swap a (the lowest frequency element) with x (the leaf at the deepest level), without increasing the cost of the tree.

Therefore, $L(\tilde{T}) \leq L(T^*)$, which means the new tree $\tilde{T}$ is also optimal.

Proving Claim #2: The Math

As long as $f_a \leq f_x$, by swapping leaf a with a leaf x at the deepest level, the cost of the tree does not increase.

Due to this fact we can swap a (the lowest frequency element) with x (the leaf at the deepest level), without increasing the cost of the tree.

Therefore, $L(\tilde{T}) \leq L(T^*)$, which means the new tree $\tilde{T}$ is also optimal.

We can similarly swap b with y and not increase the cost, to obtain another optimal tree for which the symbols a and b are siblings.

Proving Claim #2: The Math

As long as $f_a \leq f_x$, by swapping leaf a with a leaf x at the deepest level, the cost of the tree does not increase.

Due to this fact we can swap a (the lowest frequency element) with x (the leaf at the deepest level), without increasing the cost of the tree.

Therefore, $L(\tilde{T}) \leq L(T^*)$, which means the new tree $\tilde{T}$ is also optimal.

We can similarly swap b with y and not increase the cost, to obtain another optimal tree for which the symbols a and b are siblings.

Hence, a tree with the minimum cost exists that resides in $\mathcal{T}_{ab}$.

Proving Claim #2: The Math

As long as $f_a \leq f_x$, by swapping leaf a with a leaf x at the deepest level, the cost of the tree does not increase.

Due to this fact we can swap a (the lowest frequency element) with x (the leaf at the deepest level), without increasing the cost of the tree.

Therefore, $L(\tilde{T}) \leq L(T^*)$, which means the new tree $\tilde{T}$ is also optimal.

We can similarly swap b with y and not increase the cost, to obtain another optimal tree for which the symbols a and b are siblings.

Hence, a tree with the minimum cost exists that resides in $\mathcal{T}_{ab}$.

This proves Claim #2.

Conclusion of the Proof

- **Claim #2 (Exchange Argument)** guarantees that there is an optimal tree where the two lowest-frequency symbols, a and b, are siblings.
- **Claim #1 (Inductive Argument)** guarantees that Huffman's algorithm finds the best possible tree among the set of trees where a and b are siblings.
- Combining these two ideas, the tree produced by Huffman's algorithm must be an optimal tree.

This completes the inductive step, and thus the proof of correctness for Huffman's algorithm.

Q.E.D.

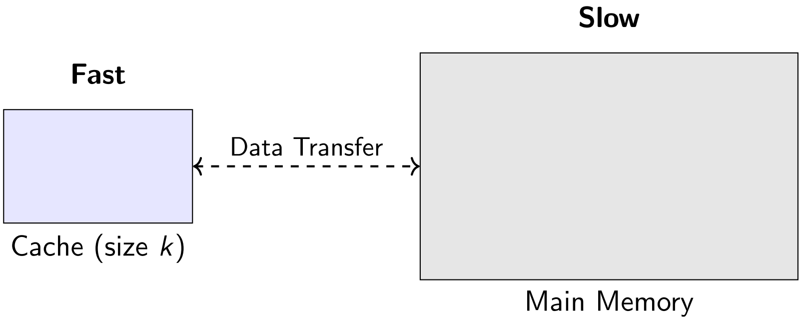
The Caching Problem

A Greedy Algorithm, Minimizing Misses

The Caching Problem

The Setup: We have a set U of n data points stored in main memory. We have a two-level memory system where we keep an extra copy of few of these data points.

- A small, fast **cache** of size k.
- A large, slow **main memory**.



The Caching Problem

The Process:

- We process a sequence of m memory requests: $d_1, d_2, \ldots, d_m$.
- If request d_i is in the cache $\rightarrow$ **Cache Hit** (fast).
- If request d_i is NOT in the cache $\rightarrow$ **Cache Miss** (slow).
  - We must fetch d_i from main memory.
  - If the cache is full, we must **evict** an item to make space.

The Goal: Design an eviction policy (a schedule) that **minimizes the total number of cache misses**.

The Caching Problem

The Process:

- We process a sequence of m memory requests: $d_1, d_2, \ldots, d_m$.
- If request d_i is in the cache $\rightarrow$ **Cache Hit** (fast).
- If request d_i is NOT in the cache $\rightarrow$ **Cache Miss** (slow).
  - We must fetch d_i from main memory.
  - If the cache is full, we must **evict** an item to make space.

The Goal: Design an eviction policy (a schedule) that **minimizes the total number of cache misses**.

Setting: We consider the "offline" setting where we have the knowledge of all future requests.

Caching: An Example (k = 3)

Let cache size $k = 3$.

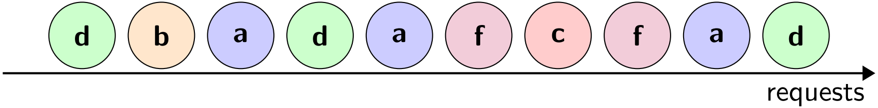
Initial cache (t=0): a b c

Caching: An Example (k = 3)

Let cache size $k = 3$.

Initial cache (t=0): a b c

Request sequence:

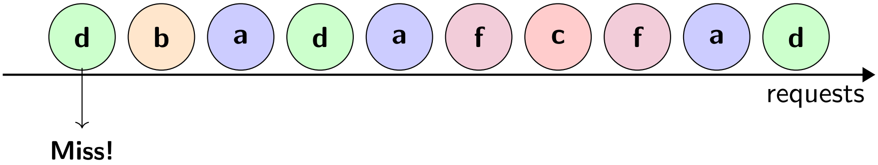


Caching: An Example (k = 3)

Let cache size $k = 3$.

Initial cache (t=0): a b c

Request sequence:



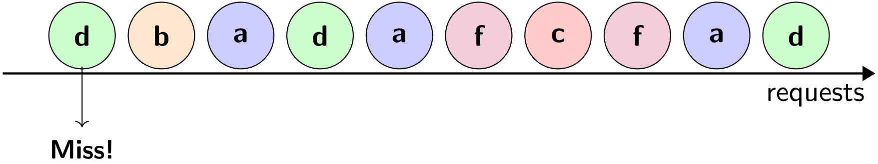
Caching: An Example ($k = 3$)

Let cache size $k = 3$.

Initial cache (t=0): a b c

cache ($t = 1$): d b c

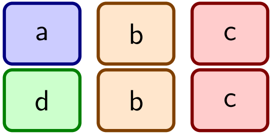
Request sequence:



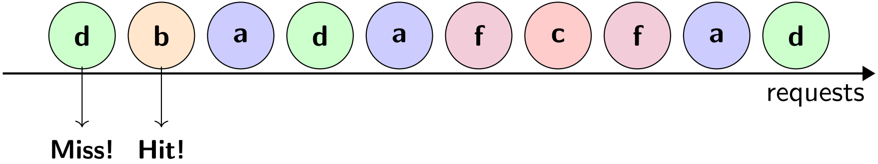
Caching: An Example (k = 3)

Let cache size $k = 3$.

Initial cache (t=0):



Request sequence:



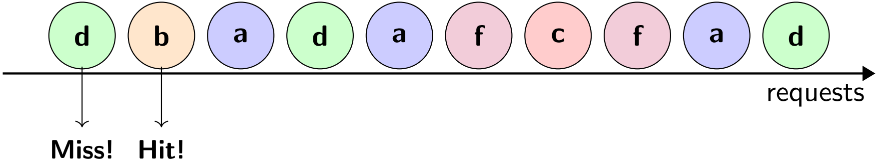
Caching: An Example (k = 3)

Let cache size $k = 3$.

Initial cache (t=0): a | b | c

cache (t = 1 & 2): d | b | c

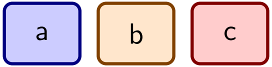
Request sequence:



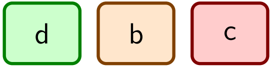
Caching: An Example (k = 3)

Let cache size $k = 3$.

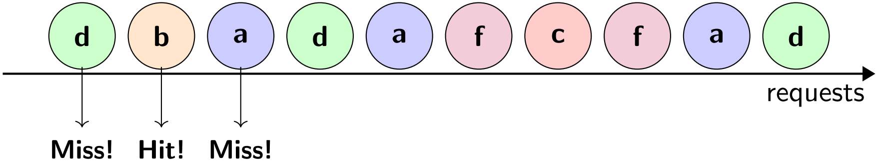
Initial cache (t=0):



cache (t = 1 & 2):



Request sequence:

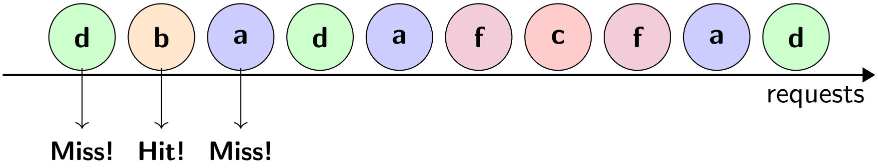


Caching: An Example ($k = 3$)

Let cache size $k = 3$.

Initial cache (t=0):	a	b	c
cache (t = 1 & 2):	d	b	c
cache (t = 3):	a	b	c

Request sequence:

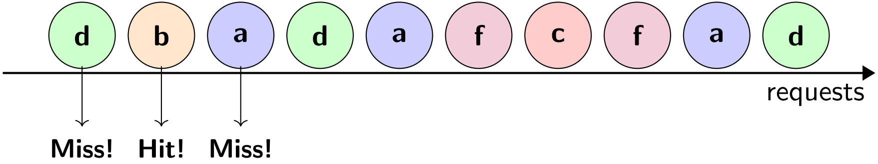


Caching: An Example (k = 3)

Let cache size $k = 3$.

Initial cache (t=0):	a	b	c
cache (t = 1 & 2):	d	b	c
cache (t = 3):	a	b	c

Request sequence:



Takeaway: Evicting a causes a refill at $t = 2$. Evicting c would have been better.

A Greedy Strategy: Evict Farthest-in-Future

Belady's Algorithm (Farthest-in-Future)

When a cache miss occurs on request d_i:

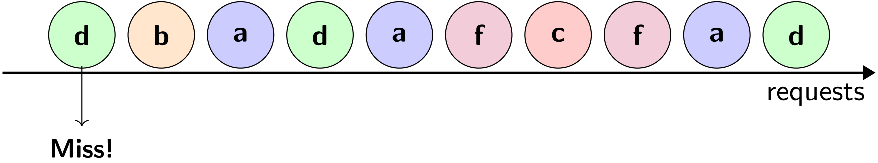
- Find the item currently in the cache that will be requested **farthest in the future**.
- Evict that item.

Intuition: Keep items that will be needed soon. Evicting an item we won't need for a long time (or ever again) seems like a safe bet.

Belady's Algorithm: Evict Farthest-in-Future

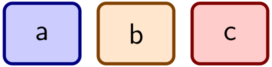
Initial cache (t=0): a b c

Request sequence:

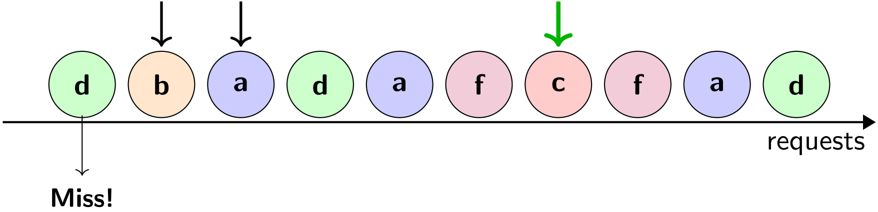


Belady's Algorithm: Evict Farthest-in-Future

Initial cache (t=0):



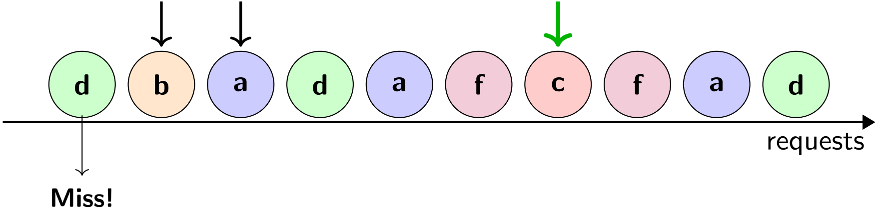
Request sequence:



Belady's Algorithm: Evict Farthest-in-Future

Initial cache (t=0):	a	b	c
cache (t = 1):	a	b	d

Request sequence:

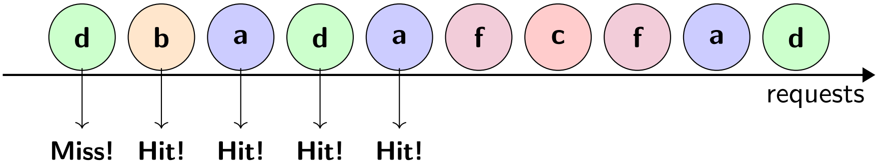


Belady's Algorithm: Evict Farthest-in-Future

Initial cache (t=0): a b c

cache (t = 1): a b d

Request sequence:



Reduced Schedules

A Canonical Form for Schedules

Reduced Schedules: A Canonical Form

Reduced schedule. Only loads an item when it is requested and not already in cache.

Reduced Schedules: A Canonical Form

Reduced schedule. Only loads an item when it is requested and not already in cache.

Does pre-loading help?

Reduced Schedules: A Canonical Form

Reduced schedule. Only loads an item when it is requested and not already in cache.

Does pre-loading help?

No! Because any schedule can be turned into a reduced one without increasing misses.

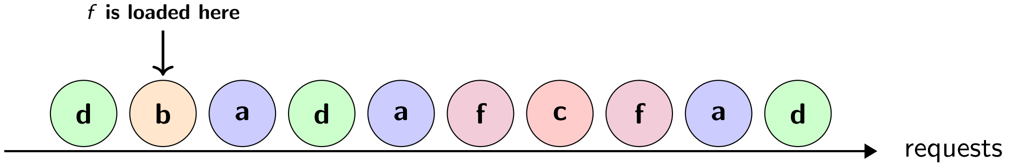
Reduced Schedules: A Canonical Form

Reduced schedule. Only loads an item when it is requested and not already in cache.

Does pre-loading help?

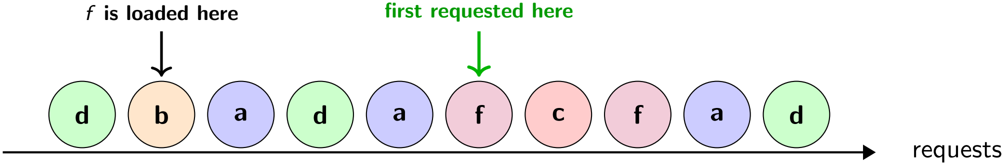
No! Because any schedule can be turned into a reduced one without increasing misses.

Suppose an algorithm *prematurely* loads an element f before any request to f. We can construct a *reduction* S' that "pretends" to load f then actually loads f only at its first request.



Reduction of a Schedule

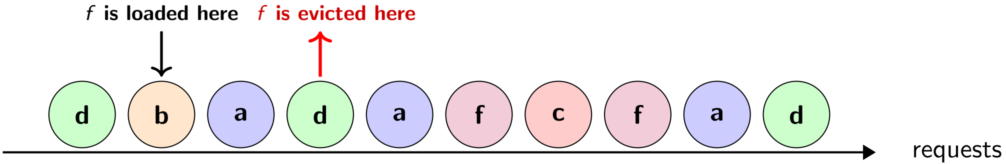
If S brings f early (when f is not requested), define S' a be to be an schedule that does not load f. S' can follow all the actions of S unless it involves f. The first event involving f is either:



Reduction of a Schedule

If S brings f early (when f is not requested), define S' a be to be an schedule that does not load f. S' can follow all the actions of S unless it involves f. The first event involving f is either:

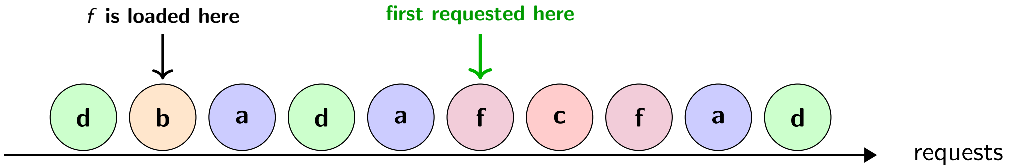
- an *eviction of f before any request* — then the early load was unnecessary and S' avoids it; or



Reduction of a Schedule

If S brings f early (when f is not requested), define S' a be to be an schedule that does not load f. S' can follow all the actions of S unless it involves f. The first event involving f is either:

- an *eviction of f before any request* — then the early load was unnecessary and S' avoids it; or
- the *first request to f* — then S' incurs the miss at that step, which we can charge to the early operation of S.

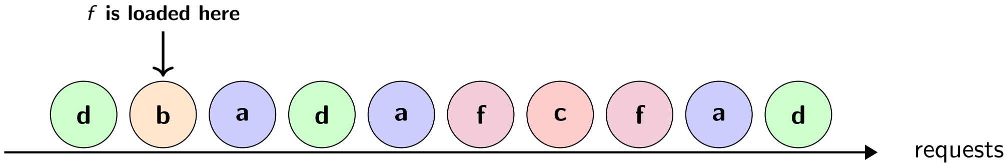


Reduction of a Schedule

If S brings f early (when f is not requested), define S' a be to be an schedule that does not load f. S' can follow all the actions of S unless it involves f. The first event involving f is either:

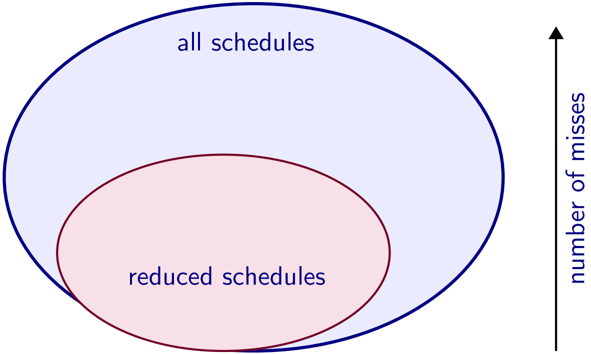
- an *eviction of f before any request* — then the early load was unnecessary and S' avoids it; or
- the *first request to f* — then S' incurs the miss at that step, which we can charge to the early operation of S.

Thus S' is reduced and brings in no more items than S; the number of misses/evictions does not increase.



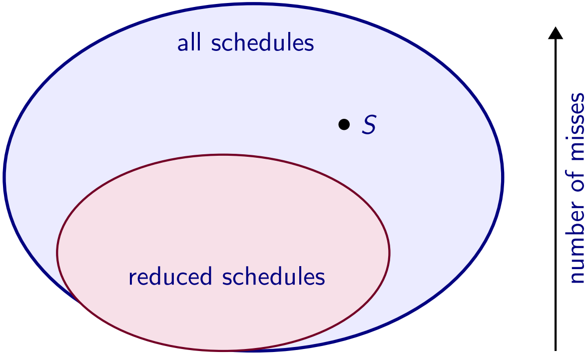
Why We Can Restrict to Reduced Schedules

Since every (possibly non-reduced) schedule can be transformed into an *equally good or better* reduced schedule, we lose no generality by restricting our attention to reduced schedules.



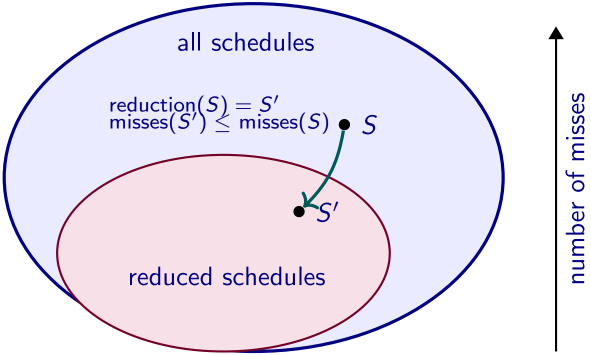
Why We Can Restrict to Reduced Schedules

Since every (possibly non-reduced) schedule can be transformed into an *equally good or better* reduced schedule, we lose no generality by restricting our attention to reduced schedules.



Why We Can Restrict to Reduced Schedules

Since every (possibly non-reduced) schedule can be transformed into an *equally good or better* reduced schedule, we lose no generality by restricting our attention to reduced schedules.



Proof of Correctness

An Exchange Argument

Proof Idea: The Exchange Argument

We will show that the Farthest-in-Future schedule (S_{FF}) is optimal.

1. Let S^* be any **reduced** and **optimal** schedule. Our goal is to show that S^* has at least as many misses as S_{FF}.

Proof Idea: The Exchange Argument

We will show that the Farthest-in-Future schedule (S_{FF}) is optimal.

1. Let S^* be any **reduced** and **optimal** schedule. Our goal is to show that S^* has at least as many misses as S_{FF}.

2. If S^* and S_{FF} are identical, we are done.

Proof Idea: The Exchange Argument

We will show that the Farthest-in-Future schedule (S_{FF}) is optimal.

1. Let S^* be any **reduced** and **optimal** schedule. Our goal is to show that S^* has at least as many misses as S_{FF}.

2. If S^* and S_{FF} are identical, we are done.

3. If not, we find the first decision where they differ.

Proof Idea: The Exchange Argument

We will show that the Farthest-in-Future schedule (S_{FF}) is optimal.

1. Let S^* be any **reduced** and **optimal** schedule. Our goal is to show that S^* has at least as many misses as S_{FF}.

2. If S^* and S_{FF} are identical, we are done.

3. If not, we find the first decision where they differ.

4. We will then perform an **exchange** to modify S^* to be more like S_{FF} **without increasing the number of misses**.

Proof Idea: The Exchange Argument

We will show that the Farthest-in-Future schedule (S_{FF}) is optimal.

1. Let S^* be any **reduced** and **optimal** schedule. Our goal is to show that S^* has at least as many misses as S_{FF}.
2. If S^* and S_{FF} are identical, we are done.
3. If not, we find the first decision where they differ.
4. We will then perform an **exchange** to modify S^* to be more like S_{FF} **without increasing the number of misses**.
5. By repeating this process, we can transform S^* into S_{FF} entirely, proving S_{FF} is optimal.

Proof Step 1: The Inversion

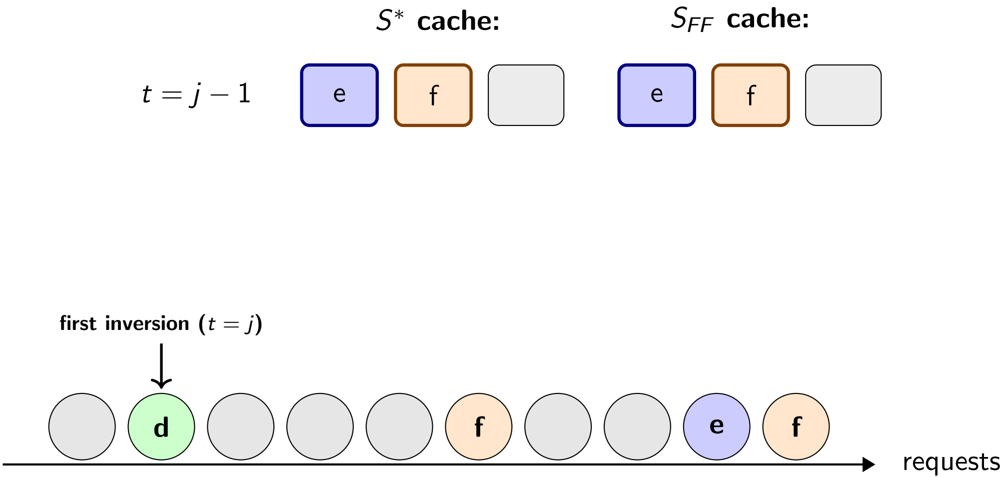
Let $d_j = d$ be the first request on which S^* and S_{FF} act differently.

- Because S^* is reduced, this happens on a cache miss for some item d.
- Before this step, both schedules have identical cache contents.
- At step j, to make room for d:
  - S_{FF} evicts e.
  - S^* evicts f.

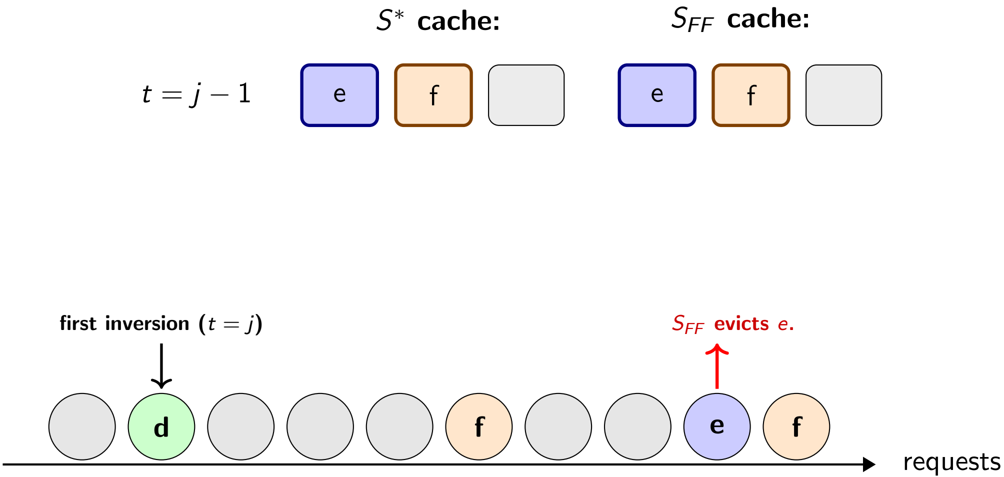
Greedy Implication

By the greedy rule of S_{FF}, e must be requested farther in the future than f.

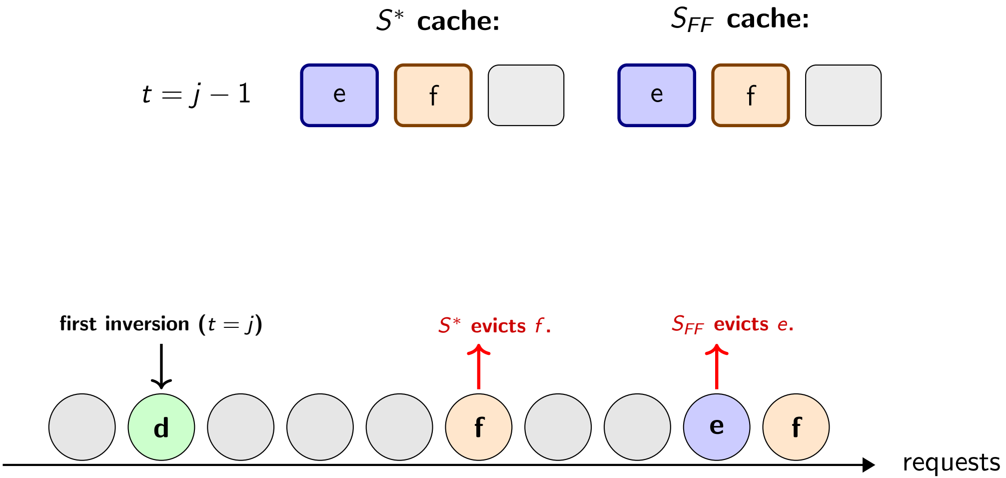
Proof Step 1: The Inversion



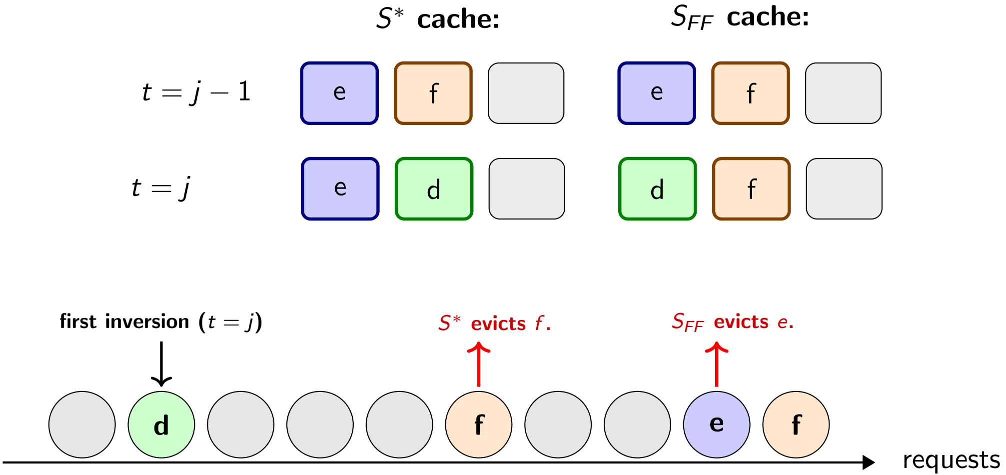
Proof Step 1: The Inversion



Proof Step 1: The Inversion



Proof Step 1: The Inversion



Proof Step 2: The Exchange

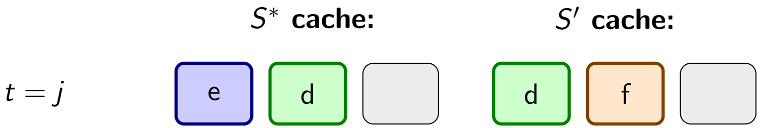
Construct a new schedule S' from S^* by changing one decision.

- Up to step j, S' behaves exactly like S^*.

Proof Step 2: The Exchange

Construct a new schedule S' from S^* by changing one decision.

- Up to step j, S' behaves exactly like S^*.
- At step j, S' evicts e (like S_{FF}) instead of f.



Proof Step 2: The Exchange

Construct a new schedule S' from S^* by changing one decision.

- Up to step j, S' behaves exactly like S^*.
- At step j, S' evicts e (like S_{FF}) instead of f.
- After step j, S' follows S^* whenever possible.

	S^* **cache:**			S' **cache:**		
$t = j$	e	d		d	f	

Proof Step 2: The Exchange

Construct a new schedule S' from S^* by changing one decision.

- Up to step j, S' behaves exactly like S^*.
- At step j, S' evicts e (like S_{FF}) instead of f.
- After step j, S' follows S^* whenever possible.
    - events that do not involve f and e.

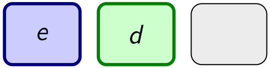
	S^* cache:			**S' cache:**		
$t = j$	e	d		d	f	

Proof Step 2: The Exchange

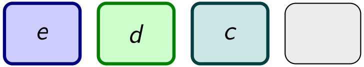
The first problematic event at $t = j'$:

S^* actions:

- S^* evicts e.

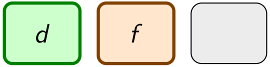


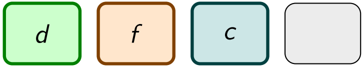
- f is requested, c is evicted.



- e is requested.

S' actions:



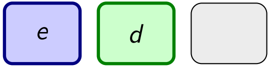


Proof Step 2: The Exchange

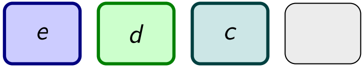
The first problematic event at $t = j'$:

S^* actions:

- S^* evicts e.



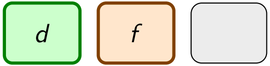
- f is requested, c is evicted.

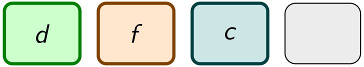


- e is requested.

S' actions:

- S_{FF} evicts f.



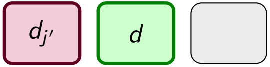


Proof Step 2: The Exchange

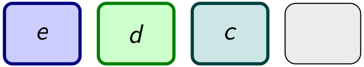
The first problematic event at $t = j'$:

S^* actions:

- S^* evicts e.



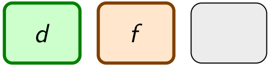
- f is requested, c is evicted.

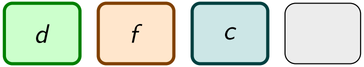


- e is requested.

S' actions:

- S_{FF} evicts f.



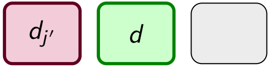


Proof Step 2: The Exchange

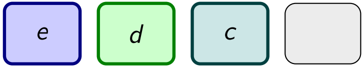
The first problematic event at $t = j'$:

S^* actions:

- S^* evicts e.



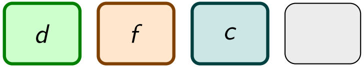
- f is requested, c is evicted.



- e is requested.

S' actions:

- S_{FF} evicts f.

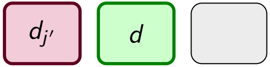


Proof Step 2: The Exchange

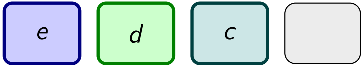
The first problematic event at $t = j'$:

S^* actions:

- S^* evicts e.



- f is requested, c is evicted.

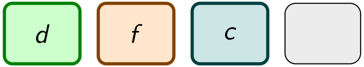


- e is requested.

S' actions:

- S_{FF} evicts f.

- Evict c, and bring back e.

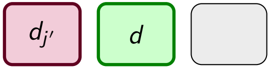


Proof Step 2: The Exchange

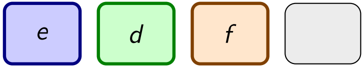
The first problematic event at $t = j'$:

S^* actions:

- S^* evicts e.



- f is requested, c is evicted.

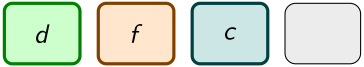


- e is requested.

S' actions:

- S_{FF} evicts f.

- Evict c, and bring back e.

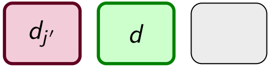


Proof Step 2: The Exchange

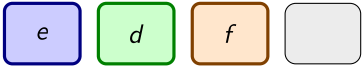
The first problematic event at $t = j'$:

S^* actions:

- S^* evicts e.



- f is requested, c is evicted.

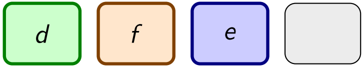


- e is requested.

S' actions:

- S_{FF} evicts f.

- Evict c, and bring back e.



Proof Step 2: The Exchange

The first problematic event at $t = j'$:

S^* actions:

- S^* evicts e.

$d_{j'}$	d	

- f is requested, c is evicted.

e	d	f	

- e is requested.

S' actions:

- S_{FF} evicts f.

d	$d_{j'}$	

- Evict c, and bring back e.

d	f	e	

- not going to happen...
  A request for f must come earlier.

Proof Step 3: Comparing Costs After the Exchange

After the exchange step, we obtained a new schedule S'.

- S' performs no more evictions than the original S^*.
- However, S' might not be *reduced*—it could load some items earlier (e.g. e) than needed.
- Let S'' be the **reduced version** of S' obtained by delaying such loads until they are first requested.

Key observation: Since the reduction never increases the number of misses,

$$\#\text{misses}(S'') \leq \#\text{misses}(S') \leq \#\text{misses}(S^*).$$

Thus S'' is also optimal and now agrees with S_{FF} through step $j+1$.

Proof Step 4: Concluding Optimality of S_{FF}

Showing that S_{FF} is optimal:

- Use **contradiction** on the maximum prefix where S^* and S_{FF} agree.
- Equivalently, prove by **induction** on the step index j.
- Repeat the exchange step until j reaches the full length of S_{FF}.

Conclusion

The **Farthest-in-Future** schedule S_{FF} is optimal.

Online Caching: Least Recently Used (LRU)

- A widely used practical strategy is **Least Recently Used (LRU)**.
- **Rule:** On a miss, evict the item whose last request was **longest ago in the past**.
- **Intuition (Locality of Reference):**
  - Recently accessed items are likely to be accessed again soon.
  - Hence the past is a useful predictor for the near future.
- Parallel with Farthest-in-Future:
  - **LRU:** Longest in the *Past*
  - **FF:** Farthest in the *Future*

The Upshot

- Farthest-in-Future is a provably optimal greedy strategy for caching.
- **Catch:** It is an **offline** algorithm requiring perfect knowledge of future requests.
- **In practice:** We use **online** heuristics; the most common is **LRU**.
- Farthest-in-Future is a crucial *benchmark* for evaluating online caching algorithms.

References

Dasgupta, S., Papadimitriou, C. H., and Vazirani, U. (2006).
Algorithms.
McGraw-Hill, Inc., USA, 1 edition.

Roughgarden, T. (2022).
Algorithms Illuminated: Omnibus Edition.
Soundlikeyourself Publishing, LLC.

Tardos, E. and Kleinberg, J. (2005).
Algorithm Design.
Pearson.